ID1021 Algoritmer och Datastrukturer
Träd

Dipsikha (Diddi) Dutta

27 September 2023

Träd

Syftet med denna inlämning är att bekanta sig hur länkade strukturer kan användas till andra typer av datastrukturer. I denna uppgift fick vi implementera tre olika huvudmetoder för klassen `BinaryTree` som huvudsakligen innehåller en `Node<K,V> root` attribut och har referenser till resten av alla noder till höger och vänster i ett trädobjekt.

För att lägga till en ny nod gör den först en kontroll i klassen `BinaryTree` för att kolla om `root` är null, och i det fallet initierar `root`.

```
public void add(K key, V value) {
    if(this.root == null) {
        this.root = new Node<>(key, value);
    } else {
        this.root.addInNodeClass(key,value);
    }
}
```

I annat fall anropar den metoden `addInNodeClass()` som initierar in noder i resten av trädet. Anledningen till att metoden är i `Node` -klassen har orsaken till att inte behöva rekursivt anropa samma metoden med en temporär nod-variabel, då `Node` -klassen fokuserar på `this` (nuvarande) nod med alla dess fyra attribut, `key, value, left` och `right`.

```
public void addInNodeClass(K key,V value) {
    if (this.key.equals(key)) {
        this.value = value ;
        return;
    }
    if (this.compareTo(key) > 0) {
        if (this.left == null) {
            this.left = new Node<>(key, value);
        } else {
```

```
            this.left.addInNodeClass(key, value);
        }
    } else {
        if (this.right == null) {
            this.right = new Node<>(key, value);
        } else {
            this.right.addInNodeClass(key, value);
        }
    }
}
```

Metoden `lookup()` har samma metodstruktur som `add()` och `addInNodeClass()` med skillnaden att den returnerar ett booleskt värde om en viss `key` finns i trädet. Returvärdet skickas om `this.key.equals(key)` är `true` och annars `false` om sökningen kommer till slutet av ett löv. (`this.left/right == null`) På samma sätt som `add()` kollar `lookup()` om `root == null` först.

```
public boolean lookup(K key) {
    if (this.root == null) {
        return false;
    } else {
        return this.root.contains(key);
    }
}
```

För att ta bort en nod i trädet gör programmet samma `root == null` check och returnerar innan den traverserar igenom hela trädet för att göra en operation. Till skillnad från de tidigare metoderna kollar `delete()` om `this.left/right.key.equals(key)` istället för om `this.key.equals(key)`. Detta för att traversera två steg åt vänster om noden som letas efter är till vänster. Och sedan ersätta den noden, med noden som ligger längs till höger om sub-trädet. På detta sätt garanteras det att ordningen i trädet inte flyttas om eller att någon nod inte försvinner med operationen. För att åstadkomma detta letas det största lövet ett steg åt vänster från den nod som ska ersättas med hjälp av metoden `findLargestLeafInLeft()`. (i vänster subt-träd) Detta returneras i noden `foundedLargestKey` och sparar den ersatta nodens båda grenar `save2right`, `save2left` innan den överskrivs med lövet. Detta sker på samma sätt spegelvänt om `this.right` är noden som ska erättas men överskrivs med högra sub-trädets minsta löv. En del kod har uteslutits pga platsutrymme men det som saknas är om det är ett löv som ska ersättas, vilket endast ändrar om den tidigare nodens referens till null.

```
public void delete(K key) {
    if (this.left.key.equals(key)) {
        Node<K,V> foundedLargestKey = this.left.left.findLargestLeafInLeft();
        if (foundedLargestKey != null) {
            Node<K, V> save2right = this.left.right;
```

```
            Node<K, V> save2left = this.left.left;
            this.left = foundedLargestKey;
            this.left.left = save2left;
            this.left.right = save2right;
        }
        return;
    }
    if (this.right.key.equals(key)) {
        Node<K,V> foundedSmallestKey = this.right.right.findSmallestLeafInRight();
        if (foundedSmallestKey != null) {
            . //samma kod som ovan men left = right
            .
        }
        return;
    }
    if (this.compareTo(key) > 0) {
        if (this.left == null) {
            return;
        } else {
            this.left.delete(key);
        }
    } else {
        . //samma kod som ovan men left = right
        .
    }
}
```

Eftersom klassen `Node<K,V>` är generisk med attributen `K key` och `V value` gör att både `BinaryTree` och `Node` har `K key` som ärver från interfacet `Comparable<K>`. Detta gör att klassen `Node` behöver implementera metoden till `Comparable<K>` `compareTo()`. Detta är till för att använda sig av javas färdiga implementation av `compareTo()` som gör att `K key` - värdena ska kunna jämföras i `add()`, `lookup()` och `remove()`.

```
@Override
public int compareTo(K otherKey) {
    return this.key.compareTo(otherKey);
}
```

I uppgiften var det specificerat att vi skulle traversera igenom trädet för att printa ut alla noder på sättet `in-order`. Detta gör att trädet printar ut värdena `V value` från minst till störst. Detta görs genom att `BinaryTree` implementerar `Iterable<>` och därmed metoden `iterator()` som returnerar ett `TreeIterator`-objekt. När trädet ska printas ut med hjälp av en `for-each-loop` använder `iterator()` sig av `hasNext()` och `next()` som avgör om stacken inte är tom respektive vilken nästa nod är. Så länge `this.next != null` (`current`) ska pekaren traversera längst till vänster och pusha alla värden på vägen och

returnera det senaste. I `next()` sparar `stack.pop()` sen senaste noden och traverserar från den ner till vänster från ett steg till höger.

```
public TreeIterator(Node<K,V> root) {
    this.stack = new Stack<>();
    pushAllLeftNodes(root);
}
private void pushAllLeftNodes(Node<K,V> current) {
    while(current != null) {
        stack.push(current);
        current = current.left;
    }
}
@Override
public Node<K,V> next() {
    Node<K,V> current = stack.pop();
    if (current != null) {
        pushAllLeftNodes(current.right);
    }
    return current;
}
```

Benchmark

Då det var lite ospecificerat vad det var vi skulle mäta i benchmarken med instruktionen "[...] compare the execution time for a growing data set.", om det är `add()` eller `lookup()` så mättes tiden för varje huvud-operation `add()`, `lookup()` och `remove()` för växande antalet noder. Kodavsnittet nedan är exempel på hur benchmarken för `lookup()` ser ut genom att genererera olika `K key`:s och annat som behövs för att mäta tiden.

```
private static float benchmarkLookup(int treeSize) {
    long t0, t1 = 0;
    float takeTime = 0;
    for (int i = 0; i < tries; i++) {
        int[] randomLookupElements = GenerateArray.unSorted(treeSize, 10000);
        BinaryTree<Integer, Integer> tree = fillTreeWithNewNodes(treeSize);
        for (int j = 0; j < treeSize; j++) {
            t0 = System.nanoTime();
            tree.lookup(randomLookupElements[j]);
            t1 = System.nanoTime();
        }
        takeTime = averageTime(t1 - t0);
    }
    return takeTime;
}
```

```java
private static BinaryTree<Integer, Integer> fillTreeWithNewNodes(int treeSize) {
    int[] randomKeys = GenerateArray.sorted(treeSize, 10000);
    int[] randomValues = GenerateArray.unSorted(treeSize, 10);
    BinaryTree<Integer, Integer> tree = new BinaryTree<>();
    for (int i = treeSize/2; i >= 0; i--) {
        tree.add(randomKeys[i], randomValues[i]); //left subtree
    }
    for (int i = treeSize/2; i < treeSize; i++) {
        tree.add(randomKeys[i], randomValues[i]); //right subtree
    }
    return tree;
}
```

Resultat

n	add()	lookup()	traverse()
100	369	0,09	20
250	369	0,10	23
500	543	0,13	43
1000	812	0,18	61
2500	1008	0,22	68
5000	1341	1,01	73
9000	2031	1,44	186

Table 1: Tid i mikrosekunder µ. Resultatet visar för tiden det tar för olika typer av operationer att exekvera med ett växande antal element

Diskussion

Så som resultatet visar är mätvärdena någorlunda stabila vilket beror på att träd-strukturer är effektiva strukturer för olika typer av operationer. Detta bygger på principen om att trädet redan är sorterad när den konstrueras då `left` alltid är mindre än föregående nod och `right` är alltid större.

Att hitta en viss nod har en tidskomplexitet på $O(log(n))$ förutsatt att trädet är balanserat. Detta avgörs mycket av om `root` är ett godtycklig värde som har ett ungefärligt mittenvärde på det intervall som data-mängden ligger på. Detta är också anledningen till att de `key` -värden som genereras i benchmarken är sorterade och börjar initiera trädet från mitten av den sorterade arrayen och fyller vänster respektive höger subträd. Detta garanterar att trädet blir balanserat eftersom att `root` är ett mittenvärde. Annars skulle det innebära att tidskomplexiteten har samma funktion som en länkad lista $O(n)$, vilket inte gör trädet effektivt längre.

Operationen `lookup()`, som också används när trädet ska lägga till en nod, har en tidskomplexitet likadan som binär-sökningen $O(log(n))$ och har en långsam utveckling ju större antalet element `n`. Men en nackdel är att traversering i träd oftast utgörs med hjälp av rekursiva algoritmer som kan vara en nackdel i java-språket då java är bäst optimerad för loopar istället för rekursiva anrop, men operationerna kan även implementeras med hjälp av loopar.

Ifall noder plockas ut från ett iterator-objekt som använder sig av en stack, och sedan lägger till noder i trädet kommer iterator-objektet inte att påverkas. Detta är för att exekveringen av `Treeiterator` sker i en `for-each` loop och körs med hjälp av `root`-objekt den har fått in i konstruktorn, vilket inte påverkar iteratorn om träd-objektet uppdateras med nya noder, därmed kommer inte några värden förloras.